REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 04Apr2003	3. REPORT TYPE AND DATES COVERED Final 01Sep2002 - 28Feb2003

4. TITLE AND SUBTITLE
Toolkit to Support Parallel Adaptive Computations on Unstructured Meshes

5. FUNDING NUMBERS
DAAD19-02-C-0080

6. AUTHOR(S)
John Tourtellott, Saurabh Tendulkar, Mark Beall, Mark Shephard

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Simmetrix Inc.
10 Halfmoon Executive Park Drive
Clifton Park, NY 12065

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
44141.1-MA-STI

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12 a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Report developed under STTR contract for topic ARMY02-T003. The initial development of a software toolkit to support parallel adaptive numerical analysis is described. The tools are focused on unstructured adaptive techniques, which are among the most difficult to implement in parallel. To support dynamic load balancing, software was developed to capture and utilize a hardware model, which represents the set of computing resources available for a given problem (analysis) at hand. Mesh refinement software was extended for parallel operation, by adding mesh migration steps that enable modifications to be made to mesh entities lying on partition boundaries. The resulting capabilities were demonstrated in a parallel adaptive Navier-Stokes analysis. A new methodology for solution transfer, which combines the update of solution information with the mesh modification, was also implemented and compared to a global solution transfer procedure used in a commercial finite element code. The results showed that the new procedure leads to smaller relative difference in strain norm. Methods for reducing the storage required for adaptively evolving meshes were also investigated. An approach based on storing differences between successive mesh iterations showed that a significant reduction in file space is feasible.

14. SUBJECT TERMS
STTR report, parallel, adaptivity, load balancing, mesh generation

15. NUMBER OF PAGES
15

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION ON THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Toolkit to Support Parallel Adaptive Computations on Unstructured Meshes

1. Distributed Mesh Environment and Dynamic Load Balancing

Adaptive computation is recognized as a necessary component of reliable scientific computation. Strategies utilize automatic refinement and coarsening of meshes and order variation guided by a posteriori estimates. Parallelization of adaptive methods is complicated by the constantly evolving nature of the domain discretization (the mesh) and associated computational loads. Even if the initial mesh was distributed for proper load balance with minimum interprocessor communication, adaptivity will introduce load imbalances and communication complications. Therefore, tools are required to maintain effective load balance. Central to these tools is the ability to automatically update the communication links as meshes evolve. These links must communicate information about mesh changes required on neighboring processors due to a change on the given processor. Since a number of operations (e.g. mesh modifications and dynamic load balancing) require the ability to modify the partitions by moving mesh and associated information between processors, a general set of mesh migration procedures are needed. To maintain load balance, dynamic load balancing procedures must operate in parallel on an already distributed mesh.

Parallel processing is being performed on everything from the largest tightly coupled supercomputers to clusters of workstations. Any adaptive strategy seeking optimal performance has to account for processor speed, memory, and communications hierarchies. Such hierarchical and heterogeneous systems are increasingly common and present additional challenges for the development of efficient dynamic load balancing procedures.

The Phase I effort focused on completing a general set of procedures to maintain appropriate load balancing of adaptive calculations. Since the resulting capabilities will be used by a wide variety of applications, a set of basic tools that can be combined in a number of different ways as appropriate for the specific application were developed. To realize this, the Phase I work concentrated on generalizing the current procedures to utilize predictive methods to load balance the processes. The most general viewpoint of this involves the use of weights to alter the partitioning so that the resulting partitions are more likely to have optimal load balancing and communication.

Weighting the resources available to perform the work must include both available processor power and communication performance. Maintaining load balance while minimizing communications when doing adaptive simulation applications is usually done using graph-based re-partitioning that allows assigning weights to both vertices and edges of the graph. Procedures to set these weights based on an a priori knowledge of the system hardware can be done via parameters in a hardware model. This should be effective when the associated computing and communication resources are dedicated to the simulation. However, in cases where the resources are shared, the speed of processor computation and communications vary with system load. In these cases it is necessary to monitor the system performance and to adjust weights in an appropriate manner at the dynamic load balancing

steps. Phase I efforts focused on the development of the hardware model and applying it with a priori hardware computational and communication parameters. Consideration of system loading will be added in Phase II.

In Phase I, therefore, the Simmetrix software for parallel mesh adaptation was extended to better leverage the heterogeneous processing power and communications capabilities found in modern computing systems using an apriori model. Now, rather than simply creating equal partitions based on the number of processors, the software can utilize weighting based on the computational environment to create partitions that are more appropriate for the available resources.

1.1 Hardware model

The upgraded partitioning software enables applications to tailor mesh partition size to each processor, based on its individual computational, storage, and network-communications capacity. To establish a basis for setting partition size, software was developed to capture a hardware model representing the set of computing resources available for a given problem (simulation) at hand. The hardware model is organized as a hierarchy of *ComputingItem* objects and *ConnectionItem* objects. ComputingItem objects represent either individual computing machines or groups of interconnected computing resources; ConnectionItem objects represent the communications paths among ComputingItem objects.

The hardware model implemented in Phase I employs a tree-based model of the partitions. The root of the tree represents the total execution environment. The children of the root node represent high level divisions well suited to reflecting different networks connected to form the total computing environment. Sub-environments like alternative SMP structures or even processor memory hierarchies are represented through the recursive subdivision of tree nodes. Each node in the tree reflects its relative computing power and percentage of load. Computing nodes at the leaves of the tree house data representing the relative computing power of their processors. Network nodes maintain bandwidth and latency properties and aggregate computing power calculated as a function of the powers of their children and the network performance.

The hardware model tree needs to be initialized at the start of a parallel computation to represent the topology of the network and identify the terminal nodes of the tree with their relevant characteristics. Ideally this process will be completely automatic, although the initial implementation was done with prespecified information. Automatic construction of the hardware model will be considered in the Phase II development.

Figure 1 is the class diagram for the ComputingItem software hierarchy. The ComputingItem class itself is an abstract base class for two concrete classes: the *ComputingGroup* class, which represents an aggregation of ComputingItem objects, and the *ComputingMachine* class, which represents a discrete computing platform. Each ComputingItem object contains one or more *Port* objects, which represent communications-access points. A typical ComputingMachine object, for example, would include one Port object to represent the network interface card in the corresponding hardware. The Port

class includes an enumeration to indicate the type of connection supported; examples include 100-Base-TX, 1000-Base-T, and a generic parallel bus.

In order to represent the complete network topology, the ComputingGroup class also contains one or more *ConnectionItem* objects and one or more *Wire* objects. The ConnectionItem object is used to represent network interconnection hardware such as routers, switches, and hubs, as well as simpler, point-to-point wiring. Wire objects, as implied by their name, are used to represent the internal connections within a ComputingGroup object. This includes wires connecting ComputingItems to ConnectionItems, ComputingItems to other ComputingItems, ComputingItems to external Ports, and ConnectionItems to external Ports.

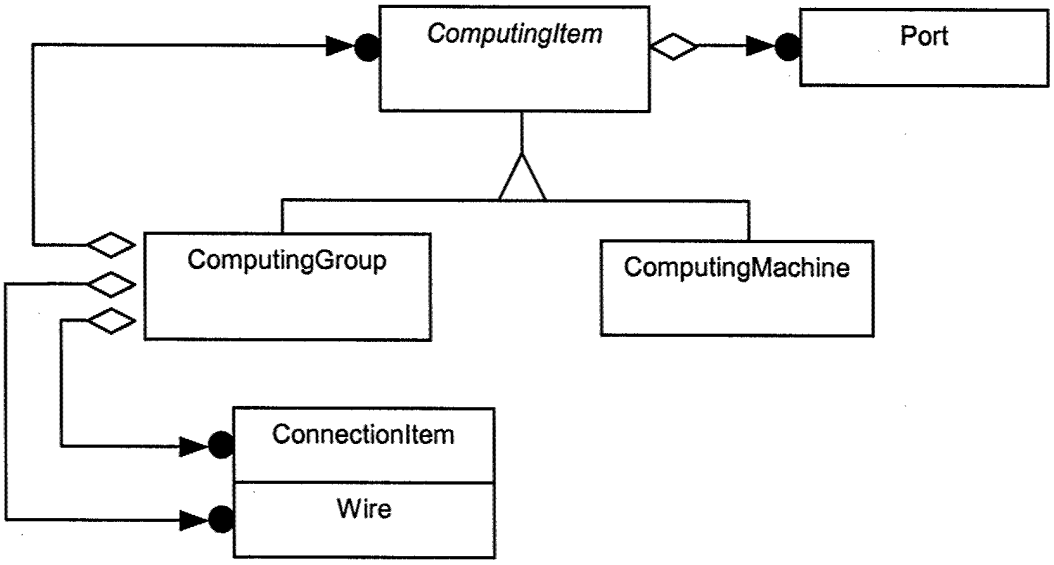


Figure 1. ComputingItem class hierarchy

Figure 2 shows a simple computing network that can be modeled using the ComputingItem classes. The top of the hardware model hierarchy is a *ComputingGroup* object, labeled "system", that represents the entire hardware model. This top-level object contains two other ComputingGroup objects, labeled "group1" and "group2". The top-level object also has a port for its uplink to an external router, and a point-to-point connection between group1 and group2.

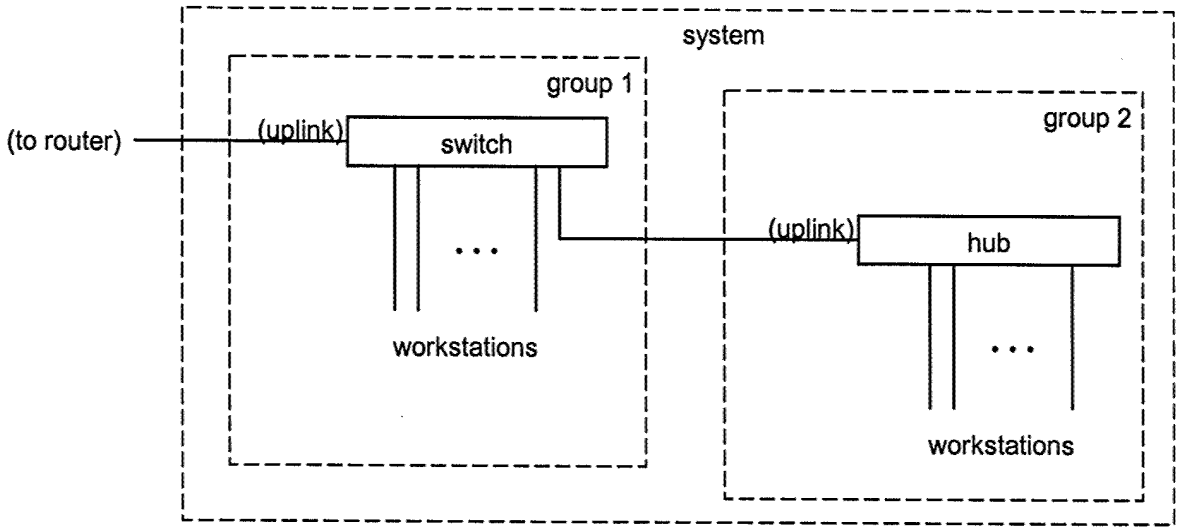


Figure 2. Example of simple computing workgroup

At the next level down, the group1 object contains a ConnectionItem object to represent the network switch and a set of ComputingMachine objects to represent computing workstations. ComputingGroup group1 also contains two external ports, one for the switch uplink (which is also the external port for the system), and the other for a switch input that is connected to group2. The group2 ComputingGroup object has one port for its uplink connection to group1, one ConnectionItem object representing a network hub, and a set of ComputingMachine objects, one for each workstation.

To complete the hardware model, each ComputingMachine object includes a representation of the resources found in the corresponding hardware platform. For each CPU in the platform, the ComputingMachine object stores one or more speed metrics (based on, for example, SPEC, LMbench, or clock speed). Each ComputingMachine object also stores the amount of physical and virtual memory in the platform, and a list of Port objects.

The ComputingItem class hierarchy was implemented and used to create hardware models for several network examples. To facilitate the instantiation of these hardware models, the software includes code to create and initialize ComputingItem objects from the contents of an XML-formatted description file. An excerpt from one of those XML files is shown in Figure 3.

```
<ComputingGroup>
   <name>system</name>
   <Port>
      <type>Port_wiring</type>
      <name>system-uplink</name>
      <wire>group1-to-system-uplink</wire>
   </Port>
   <ComputingGroup>
      <name>group1</name>
      <Port>
         <!-- This port connects group1 to group2 -->
         <type>Port_wiring</type>
         <name>group1-downlink</name>
         <wire>group1-downlink</wire>
         <wire>group1-to-group2</wire>
      </Port>
```

```
<Port>
   <!-- This port goes to (external) router -->
   <type>Port_wiring</type>
   <name>external</name>
   <wire>group1-uplink</wire>
   <wire>group1-to-system-uplink</wire>
</Port>
<ConnectionItem>
   <name>3Com 4400</name>
   <connectionType>Conn_switched</connectionType>
   <totalRate>17600</totalRate>
   <units>PortRate_Mbps</units>
   <numPorts>48</numPorts>
   <Port>
      <type>Port_100BaseTX</type>
      <name>main</name>
      <wire>sim0-to-switch</wire>
      <wire>sim7-to-switch</wire>
      <wire>sim100-to-switch</wire>
      <wire>group1-downlink</wire>
   </Port>
   <Uplink>
      <name>uplink</name>
      <type>Port_100BaseTX</type>
      <wire>group1-uplink</wire>
   </Uplink>
</ConnectionItem>
<ComputingMachine>
   <name>sim0</name>
   <serialTag>2158663617</serialTag>
   <Port>
      <type>Port_100BaseTX</type>
      <name>sim0-network</name>
      <wire>sim0-to-switch</wire>
   </Port>
   <CPUModel>
      <architecture>sparc</architecture>
      <byteWidth>4</byteWidth>
      <MIPS>360</MIPS>
      <MFLOPS>360</MFLOPS>
   </CPUModel>
   <CPUModel>
      <architecture>sparc</architecture>
      <byteWidth>4</byteWidth>
      <MIPS>360</MIPS>
      <MFLOPS>360</MFLOPS>
   </CPUModel>
   <physicalMB>512</physicalMB>
   <virtualMB>4360</virtualMB>
</ComputingMachine>
```

Figure 3. XML file used to instantiate hardware model (excerpt)

1.2 Load balancing strategy

As noted above, the mesh partitioning software can now support different partition sizes and, therefore, parallel mesh adaptation can be more optimally tailored to the specific hardware resources available. In the ideal case, the partitioning software would take advantage of the entire hardware model, i.e., not only the processing capacity of each CPU, but also the physical memory size and communications capacity of machines and connection devices. This would suggest the use of weights to control not only the size of each *partition*, but also the size of each *partition boundary*, based on the communications throughput available among different computing machines. Since current partitioning software does not have this capability, a heuristic method was developed to incorporate this

communications information into the partition weights, and to select and assign computing machines to a specific problem set. Although simplistic and sub-optimal in the general case, the method was useful for providing reasonable test cases in the Phase I research.

A partition weight is calculated for each computing machine by combining separate metrics for computational rate and communications rate. Looking at computational rate, in Phase I, CPU clock rate (in MHz) was used as the speed metric. Since the workload for a given CPU is expected to expand linearly with the size of the partition, the partition weight for a given CPU is set proportional to its speed metric.

The communications metric is more involved since it takes into account the communications paths between each pair of computing machines. The communications metric will be described using a simplified example, illustrated in Figure 4. In this example, there are three computing machines, each with the same CPU/speed metric but different ports (network interface cards) supporting 10, 100, and 1000 Mbps, respectively. To keep the example simple, the machines are interconnected by a 1000 Mbps network switch with aggregate capacity over 15,000 Mbps (so that the impact of the switch on overall communications throughput is negligible). For the machines with 10 Mbps and 100 Mbps ports, the overall communications rate is limited to the port rate. For the 1000 Mbps machine, the overall communications rate is limited to the rate at which it can communicate to the other two machines, which, for this example, is 100 plus 10, or 110 Mbps.

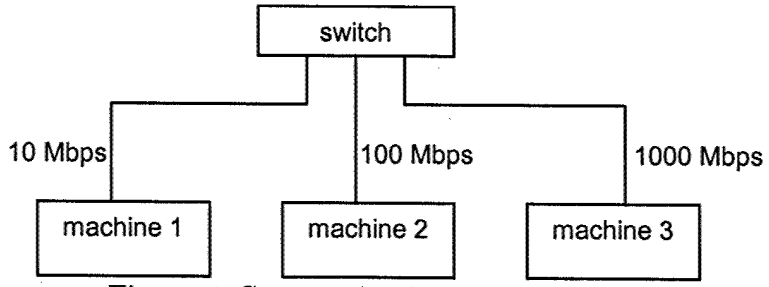


Figure 4. Communications metric example

In addition to the communications rate limit for each machine, the communications metric also takes into account the network traffic expected for a given partition. In a typical application, this would be proportional to the partition boundary size. Since there is no general way to calculate boundary size for a given partition size, the Phase I software uses a simple estimate that the boundary size is proportional to a power of the partition size. As an analogy, the surface area of a sphere is proportional to the volume of a sphere raised to the 2/3 power. The communications metric includes a *boundary factor* parameter, which can be set for each application based on user judgment and experience.

To combine the computational and communications metrics, the two values are multiplied together after raising the communications metric to a *bandwidth importance* parameter, to reflect the fact that some applications incur substantial communications overhead and others incur none. This gives the following equation for each computing machine:

$$partitionWeight = computingMetric * communicationsMetric^{\frac{bandwidthFactor}{boundaryFactor}}$$

Referring back to the example in Figure 4, the partition weights for different bandwidth factors are shown in Table 1 (based on a boundary factor of 0.5). For a bandwidth factor of zero, all three machines have the same weight, since they have the same computing metric. At a bandwidth factor of 1.0, the weights are proportional to the communications metric. Since the first machine has a network interface with much lower throughput relative to the other machines, its partition weight drops rapidly as the bandwidth factor increases.

Table 1. Partition weights for simple example

Machine	Bandwidth Factor		
	0.0	0.5	1.0
machine 1 (10 Mbps)	33.3 %	4.5 %	0.5 %
machine 2 (100 Mbps)	33.3 %	45.5 %	45.0 %
machine 3 (1000 Mbps)	33.3 %	50.0 %	54.5 %

Using this partition weighting scheme, software was written to construct the set of partition weights for the computing machines in a given hardware model (for a given boundary factor and bandwidth factor). The software selects and adds one machine at a time to the set, based on which of the remaining available machines produces the highest partition weight. (Note that the communications metrics must be recomputed each time a machine is added to the configuration.)

2. Parallel Adaptive Analysis

Ideally, parallel mesh generation and adaptation should be scalable with respect to time and memory, efficient in a parallel sense, and mesh stable. A process is considered scalable if the running time increases slowly with the number of processors, assuming the ratio of problem size to number of processors is constant. Parallel efficiency refers to how well the parallel procedure makes use of the computing resources available. A parallel procedure can be scalable but inefficient, and vice-versa. Mesh stability relates to the quality of the triangulations. If the quality degrades as the number of processors increases, the procedures are not mesh stable.

Simmetrix' existing toolkit-based mesh generation and mesh adaptation procedures have been effectively integrated with CAD representations and mesh-based analysis and error estimation procedures Interoperability technologies used in these procedures include operator driven interfaces to solid modeling, file transfer procedures, and object-based interface abstractions such as CORBA.

A key activity performed during Phase I of this project was the completion and extension of the current parallel mesh adaptation procedures to support modifications that improve the mesh, allow refinement to move nodes to the boundary, or perform mesh coarsening. These are more complex to parallelize since not all the mesh information needed for the operation may be on the processor partition. The basic parallelization of these processes is done as follows: The mesh modifications for which all information is on-processor are performed.

Those that require mesh information from other processors request that mesh information be migrated. After all on-processor modifications are complete, appropriate sets of mesh migrations are performed to allow more mesh modifications. This is repeated until the process is complete.

2.1 Parallel mesh refinement

In mesh refinement, new mesh vertices are typically created by splitting mesh edges. If a mesh edge is on the model boundary, then the new vertex will also be classified on the model boundary, however, the new vertex may not be at the right spatial location due to the geometric approximation made by the mesh. The processing of correcting the vertex location, called snapping, is accomplished by one or more mesh modifications. In parallel, these modifications would require substantial communications overhead if the affected mesh entities lay on a partition boundary.

There are three types of vertex snapping, in increasing order of complexity: direct snap, local mesh modification, and cavity remeshing. Direct snapping is used whenever it is possible to translate the (new) mesh vertex to its snap position without invalidating any elements in the mesh. In this case, snapping involves a simple change of coordinates. However, if direct snapping would cause the mesh to become invalid, then a set of local mesh modifications are attempted. The software implements seven different mesh modifications, each with several variations (edge collapse, region collapse, split collapse, edge swap, edge split, face swap, and vertex motion). Whether a particular modification is valid depends on many factors unique to each modification such as size and shape checks, correct representation of topology, model boundary, etc. The basic strategy is to try each modification in succession. If any modification succeeds, the vertex position is updated and the process repeated.

If the modification procedures fail to move the new mesh vertex to its snap position, the third step is to define a cavity around the problem vertex and remesh it using the volume mesher (which may create additional new mesh vertices on the boundary). Because remeshing a cavity requires much more computational work than either direct snapping or local mesh modification, this step is run only after all vertices have been processed using direct snapping and local modification. In some case, snapping one vertex frees others to be snapped, so that cavity remeshing can be avoided.

In parallel, snapping is not done to a vertex if it is on a partition boundary. Even if the vertex is not on the partition boundary but close to it, a modification may fail because some other mesh entity that would be modified lies on the partition boundary. Synchronizing information across partitions is not desirable, since the inter-partition communications overhead could be substantial. For vertices on the partition boundary, every modification attempt would have to be synchronized, as would the results, since some modifications may succeed on some partitions and fail on others. For vertices not on the partition boundary but near enough to affect other mesh entities on the partition boundary, the problem gets even more complicated.

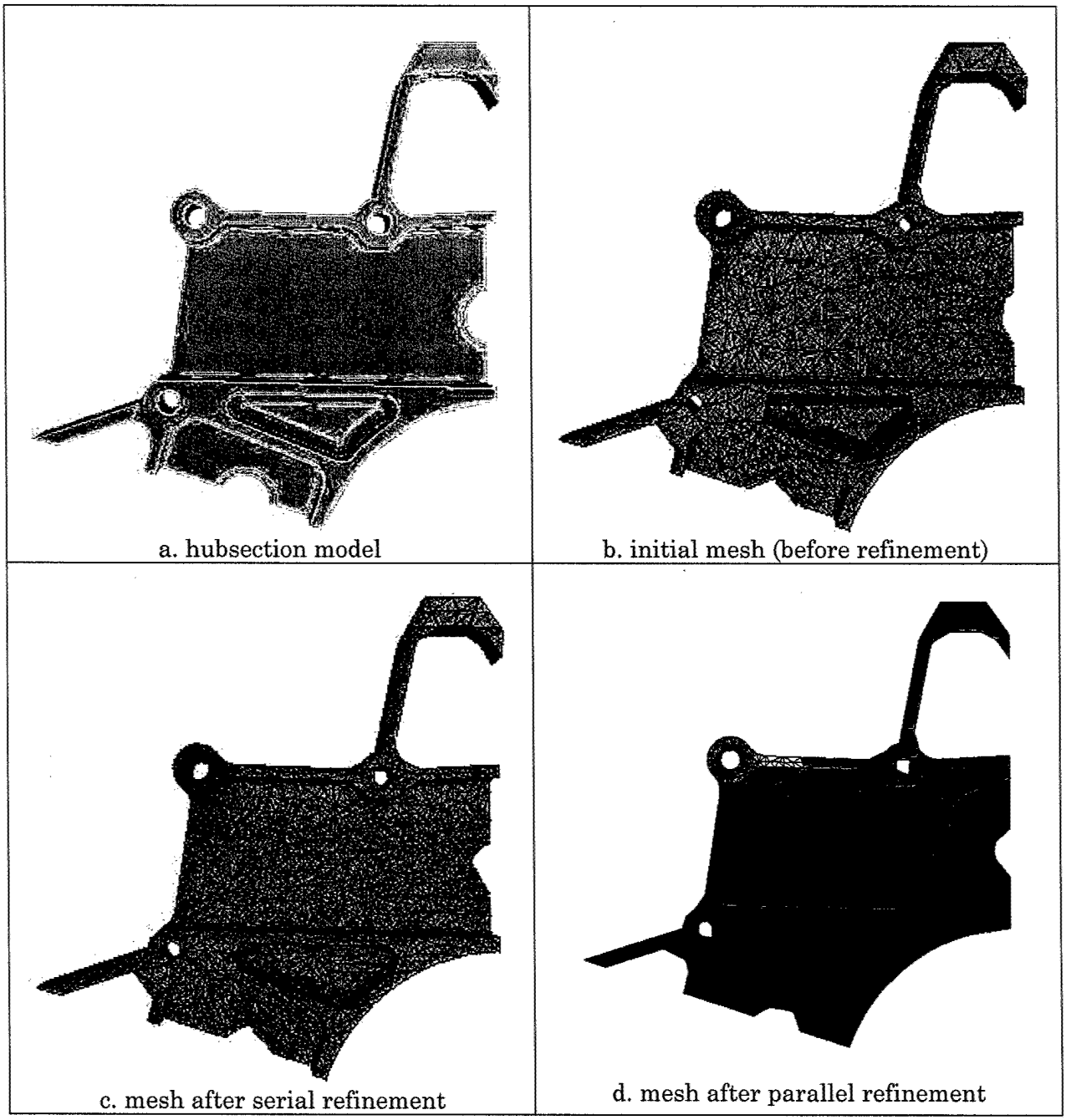
To avoid this, the parallel strategy is to repartition the mesh so that all modified entities are in the interior of one partition before snapping. The steps are:

1. Snap all vertices that can be snapped locally while making a list of vertices that cannot be snapped because they are on a partition boundary or are so near it that all modifications (and cavity remeshing) fail.
2. Move the partition boundary in a certain direction so as to get the remaining vertices in the interior. This involves migration of all regions that are affected by each vertex so that they are in one partition. The criterion as to which partition gets the set of regions for each vertex should be a converging one, so that mesh entities are not moved back and forth between repartitioning. The criterion we have chosen is to let the partition with minimum id get the regions.
3. Again snap all vertices that can be snapped locally.
4. Repeat steps 2-3 until all the partition boundary vertices are snapped. It can be shown that a maximum of three repartitioning steps are sufficient to snap all mesh vertices this way.

In doing this, various conflicts emerge in step 2 when different vertices that are to be snapped share common faces or regions. To resolve this, new software was added to aggregate these regions and arbitrate their migration among multiple partitions.

In upgrading the mesh refinement software, particularly the seven mesh modification methods, for parallel operation, a number of tradeoffs were taken into account. With single-processor refinement, a particular mesh modification can fail for geometric constraints only, whereas in parallel, a modification can fail for either geometric or partition constraints. If a modification fails due to a partition constraint, a migration step could be performed and the modification retried. The decision in Phase I was to *not* do this, but instead to proceed to the next mesh modification, trying all modifications locally before migrating mesh regions. This means that the single-processor and parallel refinement will not, in general, produce identical results.

One of the test cases for the parallel refinement software is shown in Figure 5. The geometric model for this is shown in Figure 5a, and a relatively coarse mesh of it is shown if Figure 5b. The coarse mesh in Figure 5b was the input to both single-processor and parallel versions of the mesh adapt software, with the resulting meshes shown in Figures 5c and 5d, respectively. The parallel refinement (Figure 5d) was done using three processors; the mesh entities are colored differently for each partition, using red, green, and yellow.



a. hubsection model

b. initial mesh (before refinement)

c. mesh after serial refinement

d. mesh after parallel refinement

Figure 5. Mesh refinement test case

2.2 Parallel adaptive analysis

After the load balancing software (described in Section 1) and the parallel mesh refinement software (described in Section 2.1) were implemented, the resulting capabilities were incorporated into a parallel adaptive Navier-Stokes analysis. A test was constructed involving flow over a backward-facing step. The analysis was run on two machines, with hostnames sim0 and sim100. Both machines are Sun workstations with dual CPUs. The

sim0 CPUs run at 360 MHz, and the sim0 memory bus runs at 100 MHz. The sim100 CPUs run at 750 MHz, and the sim100 memory bus runs at 150 MHz. Both equal and weighted partitions, based on the hardware model, were run on 2, 3, and, 4 CPUs.

In all, four mesh refinement iterations were run for each test, with the solver computing 25 time steps between each mesh refinement. Although the main purpose for running the analysis was to demonstrate the new parallel adaptive capability, run times were recorded for different partition configurations, and are shown in Table 2. Although a relatively simple hardware model was used, the results do show that the weighted partitioning strategy does provide a distinct advantage.

Table 2. Run time data from parallel adaptive analysis

Number of CPUs		Equal Partitions	Weighted Partitions	Relative Speedup
sim0	sim100			
1	1	1724.5	1239.4	28%
1	2	1238.4	812.4	34%
2	2	1104.0	807.3	27%

3. Parallel Transfer of Solution Fields

To have effective parallel, adaptive solution procedures it is necessary to be able to transfer solution information as the mesh is adapted. This is particularly important for transient analyses, as errors in the solution transfer process can affect the accuracy of the solution. As the meshes are modified, solution parameters must be transferred from the original mesh to the new one. The key issues that must be considered in the process are (i) preserving conservation of required quantities, (ii) maintaining accuracy, and (iii) the computational efficiency. The types of mesh modification performed can influence the complexity of the algorithms needed. The transfer process for a simple subdivision is straightforward, while for a complete remeshing it is much more complex. The solution transfer procedures developed should be capable of taking advantage of knowledge of the type of mesh modification performed when this improves the accuracy or efficiency of the process.

A promising new methodology for solution transfer combines the update of solution information with the mesh modification. Most mesh modification procedures can be expressed as a combination of a small set of local operations including splits, collapses, swaps, and vertex repositioning. Any solution update scheme that works between two meshes that are different due to a large number of these operations can also work if only one operation is applied. Thus it seems reasonable that the solution can be incrementally updated during each local mesh modification. If done in this way, the updated solution is available with little additional work after the overall mesh modification is completed. One of the significant advantages of this procedure is that no searching is required; all of the entities involved in local modification are known. Thus it is potentially a very fast operation. During Phase I, an initial version of this procedure was implemented and tested.

This local solution procedure was compared to a global solution transfer procedure used in a commercial finite element code (DEFORM). The results show that the local solution

transfer procedure leads to smaller relative difference in strain norm than the DEFORM global solution transfer procedure.

In order to compare the two solution transfer procedures, the mesh enrichment steps of a back extrusion simulation are considered. The number of elements in the original and enriched meshes for a back extrusion simulation steps are given in Table 3. To obtain the results of the DEFORM global solution transfer procedure, the original and enriched mesh, obtained by mesh modifications, are given to DEFORM explicitly, and the global solution transfer of DEFORM is used.

Table 3. Number of elements before and after mesh enrichments

Mesh Enrichment	After Simulation Step	Number of Elements in Input Mesh	Number of Elements in Output Mesh
1	4	5433	5130
2	9	5130	10723
3	46	10723	9522
4	55	9522	13466

The results obtained from the two solution transfer operations are compared by studying the relative difference in effective strain norms, which is defined by

$$E_{relative}=\frac{\left|\varepsilon\right|_{input}-\left|\varepsilon\right|_{resulted}}{\left|\varepsilon\right|_{input}}*100, \quad \text{where} \quad \left\|\varepsilon\right\|=\left[\sum_{i=1}^{nel}\int_{\Omega_i}(\varepsilon_i)^2 d\Omega\right]^{1/2}$$

In Table 4, the relative difference in effective strain norms, obtained by the two methods, are compared for some simulation steps. As can be seen from the table, local solution transfer results in less relative difference than the global solution transfer. While resulting relative differences in strain norms by local solution solutions transfer remains around 1%, relative difference by global solution transfer ranges between 5-26% for this particular simulation.

Table 4. Strain norms for local and global solution transfer procedures

Mesh Enrichment	Original Strain Norm	Strain Norm by Local Transfer	Relative Difference in Strain Norm by Local Transfer (%)	Strain Norm by Global Transfer	Relative Difference in Strain Norm by Global Transfer (%)
1	0.292136	0.291426	0.243	0.215502	26.232
2	0.929486	0.919560	1.068	0.831938	10.495
3	8.67176	8.51803	1.773	8.222860	5.177
4	10.8306	10.6794	1.396	10.217261	5.663

Effective strain distributions obtained by local and global solution transfer procedures are plotted in Figure 6. In the figure, the strain distribution of the original mesh is compared with the strain distribution obtained by the two solution transfer procedures that are

applied to an enriched (updated) mesh. Studying the figure, one can see that local solution transfer procedure results in closer strain distribution to that of original.

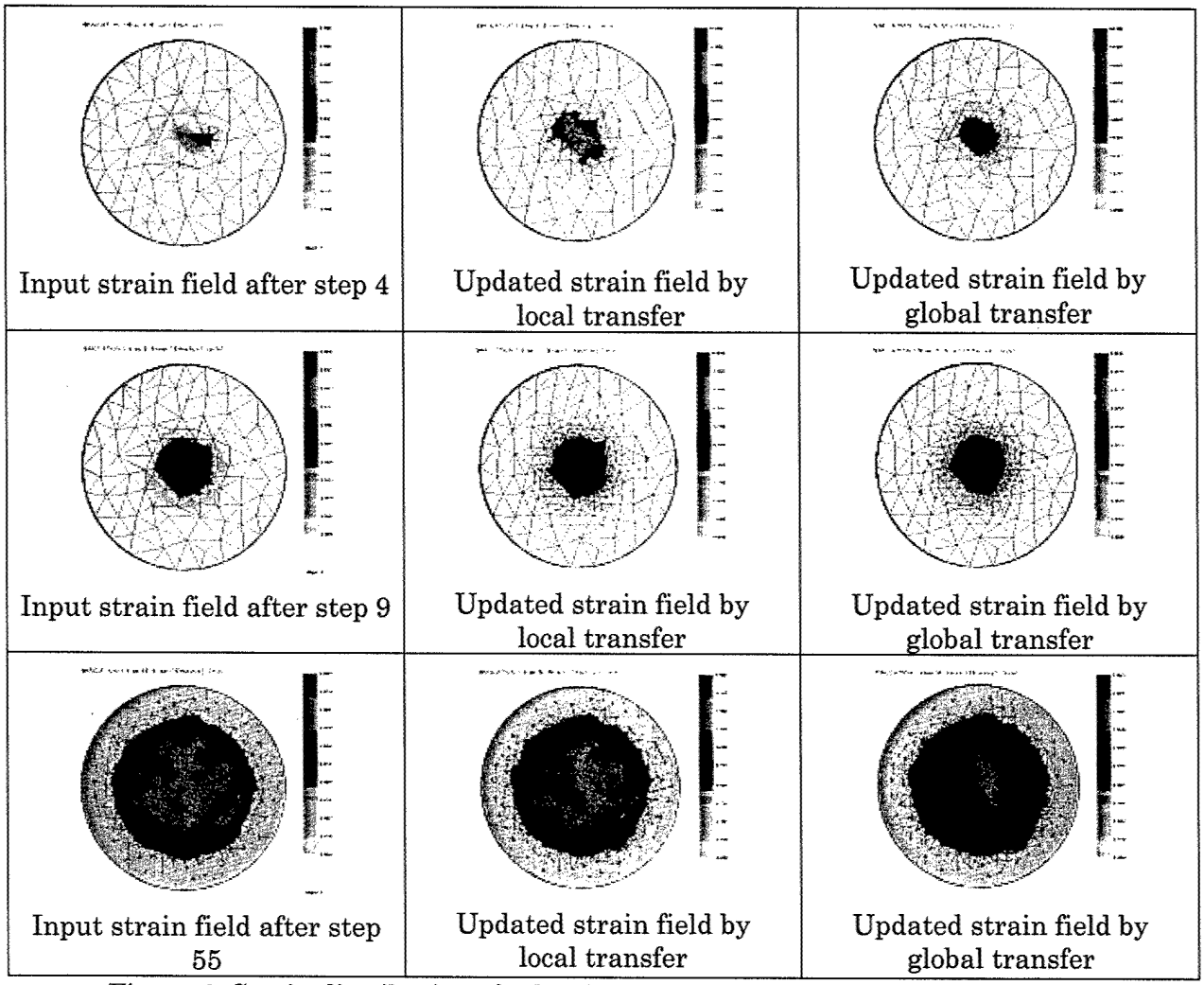


Figure 6. Strain distributions for local and global solution transfer procedures

Time spent for local and global solution transfer procedures for some steps of the simulation are given in Table 5. The number of elements in the input mesh and the updated mesh are given in Table 3. From the table, one can see that as the number of mesh modifications increase, the time spent for the local solution transfer increases. Same can be observed for global solution transfer for number of elements in the meshes. From the table, one can see that time spent for the local solution transfer is less than the global solution transfer for the first two mesh enrichment steps, and as number of mesh modifications increases time spent for local solution transfer might be more than global (as observed in mesh enrichment 3 and 4). Here, it should be noted that, the accuracy of local solution transfer is much higher than the global one (see Table 4.).

Table 5. Time spent for local and global solution transfer

Mesh Enrichments after Step	Number of Mesh Modifications	Local Transfer (sec)	Global Transfer (sec)
4	934	2.47	7.73
9	2719	5.43	11.94
46	6548	20.53	13.25
55	5307	16.33	14.57

4. Results Visualization on Adaptively Evolving Meshes

It is well known that the visualization of transient results computed on large parallel computers is a major issue due to the volume of solution data. This problem is greatly compounded in an adaptive calculation since the mesh can change thousands of times during a simulation. This may occur on hundreds of processors so that recovering the transient mesh may require loading millions of files. For example, some simulations at SCOREC run on 256 processors and require 17,000 time steps with mesh adaptation every 10 time steps, resulting in a total of 1700 meshes.

Development of actual visualization tools is outside of Simmetrix' mission and there are a number of very good groups working on these issues. Therefore, our goal is to provide support for integration with existing and future visualization tools in the structures that we are developing for supporting parallel, adaptive computation.

Over the past several years, we have been investigating the various issues related to better integration of simulation and visualization. Our current viewpoint is that one of the most important issues is that it is necessary to enable the visualization procedures to operate directly on the same data representation used for the simulation. Today, the typical means of coupling visualization and simulation requires the translation of data from the simulation representation to the visualization representation. This results in inefficiencies far beyond the obvious one of data replication. One example of this is the visualization of results that are represented by higher order basis functions. Since virtually all visualization tools support only linear interpolation, the discretization of the results transferred to the visualization tool must be much finer than the discretization used for the simulation. This can result in an explosion of the size of the data.

In Phase I, an initial investigation of effective means to represent adaptively evolving meshes with minimal storage was undertaken. The approach that seems most likely to be effective is to store only the differences between each mesh as it is adapted, similar to some of the concepts behind video compression. The Phase I investigation initially looked at simple logging of the mesh modifications as they were made, so that they could be replayed to reconstruct the mesh at each adaptation step. This approach, although very straightforward to implement, was discarded due the excessive/redundant computations that would incur during reconstruction. A simple test case was implemented by refining one mesh region in a very coarse mesh (184 regions total). Refining this one region required 150 individual mesh modification events.

A second method was investigated that, rather than logging the mesh modifications as they are computed, compares the original and refined meshes after all modifications are made. This method can take advantage of the fact that many individual modifications involve common mesh entities and only the final modifications need be stored.

Software was written to generate a list of differences between two meshes. For the same single-region refinement test, the software found that:

- Of the original 184 mesh regions, 176 were unchanged. 26 new regions were created.
- Of the original 410 mesh faces, 396 were unchanged. 55 new faces were created.
- Of the original 286 mesh edges, 280 were unchanged. 35 new edges were created.
- Of the original 61 mesh vertices, all 61 were unchanged. 6 new vertices were created.

This suggests that, for minor and/or localized mesh refinements, an incremental file, containing only the mesh elements to be added may be effective. The same differencing software was also used to analyze the four meshes generated by adaptive Navier-Stokes analysis, as described in Section 2.2. A summary of the mesh data is shown in Table 6. In this test, the number of modifications drops rapidly for each refinement step, as indicated by the number of elements row. Similarly, the number of new/changed mesh elements also drops rapidly for each refinement step. By the third refinement step (Mesh 4), the number of new elements that would have to be added to those already in memory represents only six percent of the total. These preliminary results indicate that the use of incremental files may significantly reduce the file space required to store adaptively evolving meshes.

Table 6. Mesh differences from adaptive Navier-Stokes analysis

	Mesh 1	**Mesh 2**	**Mesh 3**	**Mesh 4**
No. elements	50,187	115,001	142,985	149,867
No. new elements	-	76,693	34,800	9,010
%	-	66.7%	24.3%	6.0%

In Phase II, this work will be expanded to further address the integration of simulation and visualization in a parallel adaptive environment. As described above, the general approach will be to enable visualization tools to operate directly off of the simulation data structures that represent the mesh and the solution. One of the side effects is that this will allow very easy integration of the two capabilities for co-visualization or simulation steering.